Algoritmos y Estructuras de Datos II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

TP2: Diseño

KKOS

Integrante	LU	Correo electrónico
Burdman, Kevin	18/22	burdmankevin@gmail.com
Frau, Kenneth	189/22	kenneth.frau@gmail.com
Majic, Santiago	644/22	santiago.majic@protonmail.com
Kerbs, Octavio	64/22	octaviokerbs@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

1. Decisiones tomadas

1) Si una compra de un item deja el stock de dicho item en el puesto como 0, NO remueve el item del menu del puesto.

2)Como el TAD no lo especifica, el precio de un producto puede ser 0.

3)Asumimos que la división y, como resultado, aplicar un descuento a una compra de un Item son ambas O(1).

4)Representamos a una secuencia como una lista doblemente enlazada, de tal forma que agregar al principio/final sea O(1)

5)Se asume que copiar un natural, al igual que comparar dos naturales, es O(1).

6) Los arreglos se representan como vectores

7) Los vectores empiezan en 1

8) Se asume que modulo y division entera son O(1)

9) Se decidio incorporar una cola de prioridad a la estructura. Esta está diseñada como modulo aparte, en la que es representada como arbol binario en la que cada nodo tiene acceso al padre junto a un elemento de mayor prioridad.

10) En este TP, se usan puntero(tipo de dato) y &tipo de dato de forma indiscriminada. Ambas refieren a una referencia a una variable del tipo de dato dado. De igual forma, *variable refiere al elemento al que apunta una referencia, de manera análoga a siguiente o anterior de un iterador.

11) Decidimos no implementar la funcion "definirRapido"para el modulo DiccLog pues es similar a la ya conocida "definir"de los modulos basicos pero evitando la busqueda inicial. La complejidad es similar.

12) No ponemos como rep que el dinero > 0 para las personas que alguna vez realizaron un gasto en los puestos, para no tener la complejidad agregada de eliminar a una persona que compro de un puesto si un hackeo le deja el gasto en 0.

13) Tanto el diccLog como el conjunto se representan con AVLs, asi la complejidad de agregar o definir, pertenencia o si definido? y significado son todas de orden O(log(n))

14) Definimos para los los tipos de los heaps la relacion de orden $>_{alpha}$ segun el tipo de dato usado: por ejemplo, en el heap de gastos, se tiene como criterio un gasto mayor o un gasto igual y un id menoR. En puestos, se usa el criterio de puestoid menor, definiendo asi un min heap a partir de un max heap.
15) Tratamos de arreglar todos los errores graficos pero hay un par que fueron imposibles de solucionar, tener en cuenta este detalle.

2. Módulo Lollapatuza

Interfaz

Usa: diccLog(a, b), conj(a), puesto, idpuesto, item, cant, persona, dinero, nat, bool

Se explica con: LOLLAPATUZA

géneros: `lolla`

Operaciones:

CREARLOLLA(**in** *puestos* : `dicc(idpuesto, puesto)`, **in** *personas* : `conj(persona)`)
$\rightarrow$ *res* : `lolla`
Pre $\equiv$ {vendenAlMismoPrecio(significados(*puestos*)) $\wedge$ NoVendieronAun(significados(*personas*))
$\wedge$ $\neg\emptyset?$(*personas*) $\wedge\neg\emptyset?$(claves(*puestos*))}
Post $\equiv$ {res $=_{obs}$ CrearLolla(*puestos*, *personas*)}
Complejidad: $\mathcal{O}(\sum_{pid \in puestos}$(significado(pid,puestos) + crearPuesto(significado(pid,puestos)))
+ i + (p-1)*log(i) + a*i)
Descripción: Crea una instancia sin ventas de un Lollapatuza, en el cual los puestos de comida y las personas que pueden realizar acciones pertenecientes al TAD ya están dados como parámetros de entrada.
Aliasing: Los parametros de entrada se pasan como referencia no modificable, se devuelve una referencia modificable a res

VENDER(**in/out** lp : `lolla`, **in** pi : `puestoid`, **in** p : `persona`, **in** i : `item`, **in** c : `cant`)
Pre $\equiv$ {$p \in$ personas(lp) $\wedge$ def?(pi, puestos(l)) $\wedge_L$ $i \in$ menu(p) $\wedge_L$ haySuficiente(obtener(pi, puestos(lp)),c) $\wedge$ $lp =_{obs} lp_0$}
Post $\equiv$ {res $=_{obs}$ vender(l_0, pi, p, i, c)}
Complejidad: $\mathcal{O}(\log A + \log I + \log P)$
Descripción: Realiza una venta de un producto perteneciente a un puesto a una persona, aplicando, si existe tal, un descuento específico al producto y puesto dado, tal que la cantidad necesaria para poder aplicar ese descuento es menor a la cantidad comprada, pero es simultáneamente la mayor de las cantidades menores a la que se compró.
Aliasing: lp es una referencia modificable, tanto en entrada como en salida. El resto de los parametros se pasan por referencia no modificable.

HACKEAR(**in/out** lp : `lolla`, **in** p : `persona`, **in** i : `item`)
Pre $\equiv$ {ConsumioSinPromoEnAlgunPuesto(lp, p, i) $\wedge$ $lp =_{obs} lp_0$}
Post $\equiv$ {res $=_{obs}$ hackear(lp_0, p, i)}
Complejidad: $\mathcal{O}(\log(A) + \log(I))$
Descripción: Hackea una compra de un ítem consumido por una persona. Se hackea el puesto de menor ID en el que la persona haya consumido ese ítem sin promoción, aumentandole el stock del item en 1 y modificando el registro de la persona en sus ventas, reduciendo el gasto en el item del hackeo y modificando su compra, eliminandola de ser necesaria
Aliasing: lp se pasa como referencia modificable. El resto de los parametros de entrada se pasan como referencia no modificable.

GASTOTOTAL(**in** lp : `lolla`, **in** p : `persona`) $\rightarrow$ *res* : `dinero`
Pre $\equiv$ {$p \in$ personas(lp)}
Post $\equiv$ {$res =_{obs}$ gastoTotal(lp, p)}
Complejidad: $\mathcal{O}(\log A)$
Descripción: Obtiene el gasto total de una persona, como la suma de los gastos de la

persona en todos los puestos del lolla.
Aliasing: lp se pasa como referencia modificable, mientras que p es no modificable y res es una rcopia, ya que dinero es nat

MASGASTÓ(**in** lp: `lolla`) $\rightarrow$ res : `persona`
Pre $\equiv$ {$\neg$ $\emptyset$?(personas(lp))}
Post $\equiv$ {$res =_{obs}$ masGasto(lp)}
Complejidad: $\mathcal{O}(1)$
Descripción: Obtiene la persona que más dinero gastó en total. Si hay más de una persona que gastó el monto máximo, desempata por ID de la persona, eligiendo el de ID menor
Aliasing: Los parametros de entrada se pasan como referencia no modificable, y res es una copia, ya que persona es renombre de nat

MENORSTOCK(**in** lp: `lolla`, **in** i: `item`) $\rightarrow$ res : `puesto`
Pre $\equiv$ {$i \in$ dameUno(menu(obtenerUno($puestos$)))}
Post $\equiv$ {res $=_{obs}$ menorStock(lp, i)}
Complejidad: $\mathcal{O}(P * (\log(P) + \log(I)))$
Descripción: Obtiene el puesto de menor stock para un ítem dado. Si hay más de un puesto que tiene stock mínimo, devuelve el de menor ID.
Aliasing: Los parametros de entrada se pasan como referencia no modificable, y res es una referencia no modificable

PERSONAS(**in** lp: `lolla`) $\rightarrow$ res : `conj(persona)`
Pre $\equiv$ {True}
Post $\equiv$ {$res =_{obs}$ personas(lp)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve todas las personas que pueden formar parte en las distintas acciones del Lollapatuza
Aliasing: lp se pasa como referencia no modificable, res se devuelve como una referencia no modificable

PUESTOS(**in** lp: `lolla`) $\rightarrow$ res : `dicc(puestoid,puesto)`
Pre $\equiv$ {True}
Post $\equiv$ {$res =_{obs}$ puestos(lp)}
Complejidad: $\mathcal{O}(1)$
Descripción: Obtiene los puestos de comidas que pueden formar parte de las acciones del Lollapatuza, estando asociados cada uno con un ID único.
Aliasing: lp se pasa como referencia no modificable, res se devuelve como una referencia no modificable

Auxiliares
$>_\alpha$(**in** $personaConGasto1$: `tupla` $\langle$`dinero,persona`$\rangle$,**in** $personaConGasto2$: `tupla` $\langle$`dinero,persona`$\rangle$) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ personaConGasto1 $>_\alpha$ personaConGasto2}
Complejidad: $\mathcal{O}(1)$
Descripción: Compara dos tuplas del tipo dado y devuelve true sii la primera es mayor a la segunda según el criterio del TP
Aliasing: Los parámetros de entrada se pasan por referencia no modificable

$>_\alpha$(**in** $puestoid1$: `puestoid`,**in** $puestoid2$: `puestoid`) $\rightarrow$ res : `bool`
Pre $\equiv$ {true}
Post $\equiv$ {res $=_{obs}$ puestoid1 $>_\alpha$ puestoid2}
Complejidad: $\mathcal{O}(1)$

Descripción: Comparacion de puestoid dado el criterio de este tp, en el que un puestoid tiene mayor prioridad que otro si el primero es menor(ambos son naturales)
Aliasing: Los parámetros de entrada se pasan por referencia no modificable

Representación

Representación de Lollapatuza:

`lolla` **se representa con** `estr`

donde `estr` es `tupla`(*ColaDeGastos*: `colaPrioridad(tupla<dinero,persona>)`
, *personas*: `diccLog(persona,tupla⟨enGastos:itColaPrioridad,`
`comprasSinDescuentoPorItem:diccLog(item,`
`tupla⟨colaPrioridad(secu(compra)),diccLog(puestoid,itColaPrioridad)⟩)`
, *conjPer*: `conj(persona)`
, *puestos*: `conj(puesto))`
, *puestosPorId*: `diccLog(puestoid,puesto)`
, *precio*: `dicclog(item, nat)`)

donde **item, precio, persona, dinero, puestoid** son NAT
donde **compra** es Tupla ⟨ puntero(puesto),puntero(secu(venta)), precio, puestoid ⟩
donde **venta** es Tupla⟨item, cantidad⟩

Invariante de Representacion

1) e.conjPer = claves(e.personas).
2) e.puestos = significados(e.puestosPorId).
3) Para todo item que es clave de precio, existe un puesto que vende tal item.
4) Para todo item que es clave de precio, todos los puestos que venden ese item lo venden al significado del item en precio.
5) Además en el diccLog(puestoid,itColaPrioridad) de las compras de un item para toda persona, todos los puestosid pertenecen a claves(puestosPorId).
6) Un puesto tiene un único puestoid que lo identifica.
7) Todos los nodos de ColaDeGastos representan a una persona de personas, con su respectivo gasto total. Toda persona en personas tiene un nodo en ColaDeGastos.
8 - No hay dos nodos con la misma persona en ColaDeGastos.
9) Para toda persona registrada, el puntero enGastos apunta al nodo de la persona en ColaDeGastos (el segundo elemento de la tupla coincide).
10) Para toda persona, claves(comprasSinDescuentoPorItem) = claves(precio) (decisión: definimos todos los items para el diccionario de comprasSinDescuentoPorItem al momento de crear una instancia del Lollapatuza, para no chequear si existe el item al hacer una compra).
11) Para todo puntero a nodo de la cola de prioridad de comprasSinDescuentoPorItem de toda persona, la secuencia de compras debe tener al menos un elemento.
12) Para toda persona, en comprasSinDescuentoPorItem, un puntero es significado de un puestoid en el diccionario sii en el nodo a donde apunta, en la primera compra de la secuencia, el ultimo elemento de la compra coincide con el puestoid del puesto de la compra.
13) Toda secuencia de compras de un nodo de la cola de prioridad de comprasSinDescuentoPorItem para toda persona dada, cumple:

a)Todos los primeros elementos de cada compra apuntan al mismo puesto.
b)El puestoid, esta definido en puestosPorId, y luego, cuando se busca el significado, es el mismo puesto al que apunta el primer elemento de la tupla .

c)El segundo elemento apunta a una venta asociada al puesto y la persona dada, donde el item de la venta coincide con el item de comprasPorDescuentoSinItem, y no tiene descuento aplicado.

14) Para toda persona, su gasto total equivale a la suma de los gastos de la persona en todos los puestos del conjunto de puestos.

Rep : dic $\longrightarrow$ bool

Rep(d) $\equiv$ true $\iff$ (1 $\wedge$ 2) $\wedge_L$ (5 $\wedge$ 10) $\wedge_L$ 9 $\wedge$ 11 $\wedge_L$ 3 $\wedge$ 4 $\wedge$ 6 $\wedge$ 7 $\wedge$ 8 $\wedge$ 12 $\wedge$ 13 $\wedge$ 14

Abs : estr e $\longrightarrow$ lollapatuza {Rep(e)}

Abs(e) $\equiv$ ($\forall$ l: *lolla*) (l $=_{obs}$ e $\iff$
puestos(l) $=_{obs}$ e.puestosPorid $\wedge$
personas(l) $=_{obs}$ e.conjPer)

(1) { e.conjper = claves(e.personas) }

(2) { ($\forall$ p: puesto)(p $\in$ e.puestos $\iff$ ($\exists$ pid:puestoid)(def?(pid, e.puestosPorId) $\wedge_L$ obtener(pid,e.puestosPorId) = p)) }

(3) { ($\forall$ i: item)(i $\in$ claves(e.precios) $\iff$ ($\exists$ pid:puestoid)(def?(pid, e.puestosPorId) $\wedge_L$ i $\in$ claves(obtener(pid,e.puestosPorId).menu)))
}

(4) { ($\forall$ i:item)(def?(i,e.precios) $\Rightarrow_L$ (($\forall$ pid:puestoid)(def?(puestoid,e.puestosPorId) $\wedge_L$ i in claves((obtener(puestoid,e.puestosPorId)).menu) $\Rightarrow_L$ (obtener(i,obtener(puestoid,e.puestosPorId).menu)).precio = obtener(i,e.precios)))
}

(5) { ($\forall$ p: persona)(def?(p, personas) $\Rightarrow_L$ (($\forall$ i: item)(def?(i,obtener(p,personas).comprasSinDescuentoPorItem))) $\Rightarrow_L$ claves(π_2(obtener(i,obtener(p,personas).comprasSinDescuentoPorItem))) $\subseteq$ claves(puestosPorId)
}

(6) { ($\forall$ pid,pid':puestoid)((pid,pid' $\subseteq$ claves(e.puestosPorId) $\wedge$ pid $\neq$ pid') $\Rightarrow_L$ obtener(pid,e.puestosPorId) $\neq$ obtener(pid',e.puestosId)) }

(7) { ($\forall$ p: persona)(def?(p,e.personas) $\iff$ ($\exists$ t: tupla$\langle$dinero,persona$\rangle$)(t $\in$ colaAConjunto(e.ColaDeGastos)) $\wedge$ π_2(t) = p) }

colaAConjunto : colaPrioridad(α) $\longrightarrow$ conj(α)

colaAConjunto(c **:** *colaPrioridad*(α)**))) {** **if** vacía?(c) **then** $\emptyset$
else Ag(proximo(c),colaAConjunto(desencolar(c))) **end if** }

(8) { ($\forall$ p,p': persona)(def?(p,personas) $\wedge$ def(p',personas) $\wedge$ (p $\neq$ p' $\iff$ ((obtener(p,e.personas).enGastos) $\neq$ obtener(p',e.personas).enGastos)) }

(9) { ($\forall$ p: persona)(def?(p,personas) $\Rightarrow_L$ π_2(*(obtener(p,e.personas).enGastos)) = p)
}

(10) { ($\forall$ p:persona)(def?(p,e.personas) $\Rightarrow_L$ claves(obtener(p,e.personas).comprasSinDescuentoPorItem)) = claves(e.precio)
}

(11) { ($\forall$ p:persona)(def?(p,e.personas)) $\Rightarrow_L$ (($\forall$ i:item) (i $\in$ claves((obtener(p,e.personas)).comprasSinDescuentoPorItem)
$\Rightarrow_L$

(∀ s: secu(compra))(s ∈ colaAConjunto(
π_1(obtener(i,obtener(p,e.personas).comprasSinDescuentoPorItem)))
⇒ $_L$ ¬vacia?(s)))) }

(12) { (∀ p:persona)(def?(p,e.personas)) ⇒ $_L$ ((∀ i:item)
(i ∈ claves((obtener(p,e.personas)).comprasSinDescuentoPorItem)
⇒ $_L$
((∀ pid: puestoid)(pid ∈ claves(π_2(obtener(i, (obtener(p,e.personas)).comprasSinDescuentoPorItem)))
⇒ $_L$ π_4(prim(*obtener(pid,π_2(obtener(i, (obtener(p,e.personas)).comprasSinDescuentoPorItem)))))=
pid)))) }

(13) { (∀ p:persona)(def?(p,e.personas)) ⇒ $_L$ ((∀ i:item)
(i ∈ claves((obtener(p,e.personas)).comprasSinDescuentoPorItem)
⇒ $_L$ ((∀ pid: puestoid)(pid ∈ claves(π_2(obtener(i, (obtener(p,e.personas)).comprasSinDescuentoPorItem)))
⇒ $_L$ (((∀ c: compra)
(esta?(c,(*obtener(pid,π_2(obtener(i, (obtener(p,e.personas)).comprasSinDescuentoPorItem))))))
⇒ $_L$ (13B(c)∧13(c)∧13D(c))) ∧ (∀ c,c':compra)(está? (c,(*obtener(pid,π_2(obtener(i,
(obtener(p,e.personas)).comprasSinDescuentoPorItem))))) ∧ está?(c', (*obtener(pid,π_2(obtener(i,
(obtener(p,e.personas)).comprasSinDescuentoPorItem))))) ∧ c ≠ c' ⇒ $_L$ (13A(c,c'))))))))
}

(13A) : c:compra × c':compra ⟶ bool
{c y c' son compras distintas y estan definidas en estr para una misma persona, un mismo item, un mismo puestoId}

(13A) { *(π_1(c)) = *(π_1(c')) }

(13B) : e:estr × pid:puestoid × c:compra ⟶ bool
{pid es un puestoid valido de estr, la compra existe y corresponde a una persona e item dado, obtenida con el pid dado}

(13B) { def?(pid,e.puestosPorId) ∧$_L$ obtener(pid,e.puestosPorId) = *(π_1(c)) ∧ pid =
π_4(c) }

(13C) : e:estr × p:persona × i:item × pid:puestoid × c:compra ⟶ bool
{pid es un puestoid valido de estr, la compra c existe en estr y corresponde a la persona, item dado y pid dado}

(13C) { esta?(*(π_2(c)),(obtener(p,obtener(pid,e.puestosPorId).ventasPorPersona).ventas))
∧$_L$ π_3(c) = obtener(i,e.precio,π_2(*(π_2(c)))) }

(14) { (∀ a: persona)(def?(a, e.personas) ⇒ $_L$ (∀ p: puesto)(
$\sum_{p \in claves(e.puestosPorid)}$ (**if** def?(obtener(obtener(e.puestosPorid, p).ventasPorPersona,
a))
then obtener(obtener(e.puestosPorid, p).ventasPorPersona, a).gastoTotal **else** 0)
= obtener(e.personas, a).enGastos.dinero) }

Algoritmos

---

iCREARLOLLA(**in** *puestos* : `dicc(idpuesto, puesto)`, **in** *personas* : `conj(persona)`) → *res* : *lolla*

1: $res \leftarrow \langle$Vacio(),Vacio(),Vacio(),Vacio(),Vacio(),Vacio()$\rangle$
2: it ← crearIt(puestos)
3: **while** haySiguiente(it) **do**
4: viejoPuesto ← significado(puestos,siguiente(it))
5: nuevoPuesto ← crearPuesto(precio(viejoPuesto),stock(viejoPuesto),descuentos(viejoPuesto))
6: definirRapido(res.puestosPorId,siguiente(it),nuevoPuesto)
7: AgregarRapido(res.puestos,nuevoPuesto)
8: itItem ← crearIt(menu(viejoPuesto))
9: **while** haySiguiente(itItem) **do**
10: definir(res.precio,siguiente(itItem),precio(viejoPuesto,siguiente(itItem)))
11: Avanzar(itItem)
12: **end while**
13: Avanzar(it)
14: **end while**
15: itper ← CrearIt(personas)
16: **while** haySiguiente(itper) **do**
17: AgregarRapido(res.conjPer,siguiente(itper))
18: definirRapido(res.personas,siguiente(itper),⟨nullptr,Vacio()⟩)
19: dineroGastado ← ⟨0,siguiente(itper)⟩
20: punteroEnGasto ← encolar(res.ColaDeGastos, siguiente(itper))
21: persona ← significado(res.personas,siguiente(itper))
22: persona.enGastos ← punteroEnGasto
23: itItem2 ← crearIt(res.precio)
24: **while** haySiguiente(itItem2) **do**
25: definirRapido(persona.comprasSinDescuentoPorItem, siguiente(itItem2),⟨Vacia(),Vacia()⟩)
26: Avanzar(itItem2)
27: **end while**
28: Avanzar(itper)
29: **end while**
30: return res

Complejidad:$\mathcal{O}(\sum_{pid \in puestos}$(significado(pid,puestos) + crearPuesto(significado(pid,puestos))) + i + (p-1)*log(i) + a*i)

Justificacion:Como no sabemos como está diseñado el diccionario puestos, es la suma del significado para todo puesto, ya que debemos buscarlo primero para luego definirlo. Crear puesto es $\mathcal{O}$ ($\sum_{i' \in i}$(significado(i,i') + significado(s,i') + definido? (d,i') + significado(d,i') + $\sum_{d' \in d}$(significado(d,d') + d') + $MaximaCantConDescuento$) + #claves(p)). Definir rapido es O(log(n)), con $0 < n \leq p$, lo que se aproxima, al sumar los logaritmos, a p (Stirling). Agregar rapido es O(log(n)) pues es un conjunto logaritmico. Como las claves del conjunto empiezan en 0 y se agregan progresivamente elementos. Agregar para todo puesto es O(p) (Stirling). Para toda persona en personas, se agrega al conjunto de personas y se define en el diccionario de personas. Como agregar/definir es O(log(n)), con $0 < n \leq a$, por Stirling, esto es O(a), pero tambien para cada persona definimos automaticamente todos los items para comprasSinDescuentoPorItem con todo vacio, lo cual, usando la misma logica, es O(i) para toda persona, es decir, O(a*i). En peor caso, todos los puestos tienen el mismo menu, que consiste de todos los items posibles. En este caso, defino en precio 1 vez para cada item, lo cual, por Stirling, es O(i), y para todo puesto restante, debemos redefinir todos los items con el mismo precio(por precondicion), por lo que, ahora con todos los items, la complejidad para definir eo O(log(i)). Por ende, esta parte tiene complejidad O((p-1)*(log(i))+i) Toda otra operacion es O(1).

---

iVENDER(**in/out** e : `estr`, **in** pi : `puestoid`, **in** a : `persona`, **in** i : `item`, **in** c : `cant`)

```
1: puesto ← significado(e.puestosPorId, pi)
2: per ← significado(e.personas, a)
3: nuevaCompra ← AgregarVenta(puesto,per,i,c)
4: *(per.enGastos) ← *(per.enGastos) + π3(nuevaCompra)
5: siftUp(e.ColaDeGastos,*(per.enGastos))
6: if π1(nuevaCompra) ≠ nullptr then (Si es nullptr, la compra es con descuento)
7:     nuevaCompraCopia ← ⟨π1(nuevaCompra), π2(nuevaCompra), π3(nuevaCompra),pi⟩
8:     itemSinDescuento ← significado(per.comprasSinDescuentoPorItem,i)
9:     if definido?(π2(itemSinDescuento),pi) then
10:        (AgregarAdelante(significado(π2(itemSinDescuento),pi),nuevaCompraCopia))
11:    else
12:        it ← encolar(π1(itemSinDescuento),AgregarAdelante(nuevaCompraCopia,Vacia()))
13:        definirRapido(π2(itemSinDescuento),pi,it)
14:    end if
15: end if
```

Complejidad:$\mathcal{O}(\log(P) + \log(I) + \log(A))$

Justificacion: Primero obtenemos la persona del diccionario logarítmico de personas, y el puesto del diccionario logarítmico de puestos ($\mathcal{O}(\log(P)+ \log(A))$). Llama a AgregarVenta, cuya complejidad en peor caso es de ($\mathcal{O}(\log(I)+ \log(A))$). Para actualizar el gasto de la persona accede al puntero a la cola de prioridad y hace un SiftUp ($\mathcal{O}(\log(P))$) y, en el caso de que la compra no tenga descuentos, el peor caso es que tenga que definir un nuevo puesto en el que la compra del item no tenga descuento(en otro caso busco el puntero al nodo en la cola de prioridad en el diccionario y agrega adelante a la lista), ya que busca el elemento en el diccionario y no lo encuentra, lo encola en la cola de prioridad y inserta un puntero al nodo de dicha cola en el diccLog. El orden de esto ultimo es O(3*log(P)) = O(log(P)). Sumando todas las complejidades nos queda esta complejidad

iHACKEAR(**in/out** e : `estr`, **in** a : `persona`, **in** i : `item`)

1: per ← significado(e.personas,a)
2: precioDelItem ← buscar(e.precio,item)
3: ComprasSinDescuento ← significado(per.comprasSinDescuentoPorItem,i)
4: PuestoDeCompraABorrar ← proximo(ComprasSinDescuento)
5: it ← crearIt(PuestoDeCompraABorrar)
6: compra ← Siguiente(it)
7: pid ← π_4(compra)
8: compraModificada ← EliminarVentaSinDescuento(π_1(compra),a,i,π_2(compra,precioDelItem))
9: **if** π_1(compra) $\neq$ nullptr **then**
10: compra ← $\langle \pi_1$(compraModificada),π_2(compraModificada),π_3(compraModificada),pid$\rangle$
11: **else**
12: eliminarSiguiente(it)
13: **end if**
14: **if** ¬(haySiguiente(it)) **then**
15: desencolar(π_1(ComprasSinDescuento))
16: borrar(π_2(ComprasSinDescuento), pid)
17: **end if**
18: precioDelItem ← buscar(e.precio,item)
19: gastoDePersona ← *(per.enGastos)
20: gastoDePersona ← gastoDePersona - precioDelItem
21: siftDown(ColaDeGastos,gastoDePersona)

Complejidad: $\mathcal{O}(\log(P) + \log(A) + log(I))$
Justificacion: La mayoria de operaciones del algoritmo refiere a manipulación y eliminacion de punteros, todos $\mathcal{O}(1)$. Las unicas operaciones que superan esto refieren a busquedas en diccLogs, en donde la cantidad de claves están acotadas por la cantidad de items, de puestos o de personas, e insercion o eliminacion de elementos en colas de prioridad, donde la cantidad de claves tambien esta acotada por la cantidad de personas o de puestos. Esto cumple la complejidad porque, en el caso de que un puesto no sea hackeable luego de la funcion, remueve el elemento de mayor prioridad del heap ($\mathcal{O}(\log(p))$) y esa es la unica operacion en la que tengo que comparar varios puestos, y debo sumar la complejidad de las operaciones de los otros casos, que, como consisten de busquedas en AVLs de Personas o items y remover el maximo de un heap de personas, es O(log(P)+log(A))

---

iGASTOTOTAL(**in/out** e : `estr`, **in** a : `persona`) → res : dinero

1: $infoPersona \leftarrow significado(e.personas, a)$
2: $itAlGasto \leftarrow infoPersona.enGastos$
3: $res \leftarrow \pi_1(*itAlGasto)$
4: return res

Complejidad: $\mathcal{O}(\log A)$
Justificacion: Para buscar el gasto total buscamos a la persona en el diccionario de la estructura y accedemos a su gasto total en la tupla, $\mathcal{O}(\log A)$ + $\mathcal{O}(1)$ siendo A la cantidad de personas. Luego acceder a la cola de prioridad con el puntero al espacio de memoria que apunta a ella es $\mathcal{O}(1)$.

---

iMASGASTÓ(**in/out** l : `lolla`) → res : persona

1: res ← π_2(proximo(e.ColaDeGastos))

Complejidad: $\mathcal{O}(1)$
Justificacion: Accedo al próximo de "colaDeGastos" en la estructura, y eso cuesta O(1).

---

iMENORSTOCK(**in/out** e : `estr`,**in** i : `item`) $\rightarrow$ res : puesto

1: it $\leftarrow$ crearIt(e.puestosPorId)
2: puestoActual $\leftarrow$ significado(e.puestosPorID,siguiente(it))
3: minStock $\leftarrow$ puestoActual
4: idMin $\leftarrow$ siguiente(it)
5: Avanzar(it)
6: **while** haySiguiente(it) **do**
7: **if** definido?(puestoActual.menu,i) **then**
8: stockDeItem $\leftarrow$ Stock(puestoActual,i)
9: **if** ($\neg$definido?(minStock.menu,i) $\wedge$ $\neg$definido?(puestoActual.menu,i) $\wedge$
10: idMin $>$ siguiente(it)) $\vee_L$ ($\neg$definido?(minStock.menu,i) $\wedge$ definido?(puestoActual.menu,i)) $\vee_L$ (definido?(minStock.menu,i) $\wedge$ (Stock(puestoActual,i) $>$ Stock(minStock,i) $\vee$ ((Stock(minStock,i)) = Stock(puestoActual,i) $\wedge$ idMin $>$ siguiente(it)))) **then**
11: $minStock \leftarrow$ puestoActual
12: $idMin \leftarrow$ siguiente(it)
13: **end if**
14: **end if**
15: Avanzar(it)
16: **end while**
17: res $\leftarrow$ minStock
18: return res

Complejidad: $\mathcal{O}(P * (\log(P) + \log(I)))$

Justificacion: Fuera de definir y avanzar iteradores (O(1)), para todo puesto definido, debo primero ver si el item existe en el menú (O(log(I)), y luego ver si esta el item en el menu del minimo anterior (O(log(I)), y en caso de estar definido en ambos, comparar el Stock y, en peor caso, tambien el id, ambos de acceso O(1) una vez que accedemos al puesto. Como tambien tenemeos que buscar el puesto en puestosPorID, esto suma O(log(P)) a la complejidad de cada iteracion. Como se debe hacer esto para todo puesto, nos queda como complejidad $\mathcal{O}(P * (\log(P) + \log(I)))$

---

iPERSONAS(**in/out** e : `estr`) $\rightarrow$ res : conj(persona)

1: $res \leftarrow e.personas$

Complejidad: $\mathcal{O}(1)$

Justificacion: Devuelve una referencia no modificable de personas

---

iPUESTOS(**in/out** e : `estr`) $\rightarrow$ res : dicc(puestoid,puesto)

1: $res \leftarrow e.puestosPorId$

Complejidad: $\mathcal{O}(1)$

Justificacion: Devuelve una referencia no modificable de puestosPorId

Auxiliares

$>_\alpha$(**in** *personaConGasto*1 : `tupla⟨dinero,persona⟩`,
in *personaConGasto*2 : `tupla⟨dinero,persona⟩`) $\rightarrow$ *res* : bool

1: res $\leftarrow$ personaConGasto1.dinero $>$ personaConGasto2.dinero $\vee$ (personaConGasto1.dinero $=$ personaConGasto2.dinero $\wedge$ personaConGasto1.persona $<$ personaConGasto2.persona)
2: return res
Complejidad: $\mathcal{O}(1)$
Justificacion: En peor caso, el gasto es igual y se debe comparar a las personas(de tipo nat). Como son dos comparaciones de naturales, asumidas $\mathcal{O}(1)$, esto tiene complejidad $\mathcal{O}(1)$.

$>_\alpha$(**in** *puestoid*1 : `puestoid`, **in** *puestoid*2 : `puestoid`) $\rightarrow$ *res* : bool

1: res $\leftarrow$ puestoid1 $<$ puestoid2
Complejidad: $\mathcal{O}(1)$
Justificacion: Es una comparacion de naturales, asumida O(1)

---

3. Módulo Puesto de Comida

Interfaz

Usa: diccLog(a, b), item, cant, nat, puesto, dinero, persona, tupla, precio, bool

Se explica con: PuestoDeComida

géneros: `puesto`

Operaciones:

CREARPUESTO(**in** p : `dicc(item, nat)`, **in** s : `dicc(item, nat)`,**in** d : `dicc(item, dicc(cant, nat))`) $\rightarrow$ res : `puesto`
Pre $\equiv$ {claves(p) = claves(s) $\wedge$ claves(d) $\subseteq$ claves(p)}
Post $\equiv$ {$res =_{obs}$ crearPuesto(p,s,d)}
Complejidad: O($\sum_{i' \in i}$(significado(i,i') + significado(s,i') + definido?(d,i') + significado(d,i') + $\sum_{d' \in d}$(significado(d,d') + d') + $MaximaCantConDescuento$) + #claves(p))
Descripción: Inicializa un nuevo puesto, con un menú con precios, stock de productos y set de descuentos asociados a una cantidad mínima comprada de un productos predefinidos, ya que son todos pasados como parámetros de entrada.
Aliasing: Los parametros de entrada se pasan como referencia no modificable, se devuelve una referencia modificable a res

STOCK(**in** p: `puesto`, **in** i: `item`) $\rightarrow$ res : `cant`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {$res =_{obs}$ stock(p,i)}
Complejidad: $\mathcal{O}$(log(I)
Descripción: Obtiene el stock de un ítem de un puesto. El ítem debe existir en el menú de dicho puesto.
Aliasing: Devuelve una referencia modificable al stock de un determinado puesto, esto se ve afectado al hackear en Lollapatuza y al recibir una compra en lollapatuza

DESCUENTO(**in** p: `puesto`, **in** i: `item`, **in** c: `cant`) $\rightarrow$ res : `nat`
Pre $\equiv$ {i $\in$ menu(p)}
Post $\equiv$ {$res =_{obs}$ descuento(p, i, c)}
Complejidad: $\mathcal{O}(\log I)$
Descripción: Obtiene el descuento de un ítem dada la cantidad del mismo, si tal descuento existe. El descuento cumple que se aplica a una cantidad menor o igual que la cantidad dada, pero no hay otro descuento aplicado a una cantidad menor o igual a cant pero mayor al del resultado.
Aliasing: Los parámetros de entrada se pasan como referencia no modificable, y res es natural, por lo que se pasa por copia

GASTODE(**in** p: `puesto`, **in** a: `persona`) $\rightarrow$ res : `dinero`
Pre $\equiv$ {True}
Post $\equiv$ {$res =_{obs}$ precio(p, a)}
Complejidad: $\mathcal{O}(\log A)$
Descripción: Obtiene el gasto realizado por una persona en el puesto.
Aliasing: Los parámetros de entrada se pasan como referencia no modificable, y res es natural, por lo que se pasa por copia

Auxiliares

AgregarVenta(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** c: `cantidad`) $\rightarrow$ res : `tupla`$\langle$`itDiccLog,itLista,nat`$\rangle$

Pre $\equiv \{i \in \text{menu}(p) \wedge_L \text{haySuficiente}(p, i, c) \wedge p =_{obs} p_0\}$
Post $\equiv \{p =_{obs} \text{vender}(p, a, i, c) \wedge (\pi_1(\text{res}) \neq \text{nullptr}) \Rightarrow_L (\text{siguiente}(\pi_1(\text{res})) = p \wedge \pi_1(\text{siguiente}(\pi_2(\text{res}))) = i \wedge \pi_2(\text{siguiente}(\pi_2(\text{res}))) = c \wedge \pi_3(\text{res}) = \text{AplicarDescuento(i*c,descuento(p,i,c)))}\}$
Complejidad: $\mathcal{O}(\log(\text{A}))$
Descripción: Vende la cantidad dada del producto que debe encontrarse en el menu del puesto dado, aplicando el descuento si tal existe y removiendo la cantidad dada del stock. Si el stock queda con 0, no remueve tal item del menu. En caso de ser una venta sin descuento, devuelve iteradores hacia el puesto y la compra registrada y el 3er elemento de la tupla es el precio de la compra dada.
Aliasing: El puesto es pasado como referencia modificable, mientras que los restantes parámetros son pasados como referencias no modificables. Res consiste de dos iteradores, los cuales ambos son constantes, pero el iterador al puesto permite modificar el puesto y el de la compra permite modificar la compra, pero solo en caso de que la compra sea sin descuento

EliminarVentaSinDescuento(**in/out** p: `puesto`, **in** a: `persona`, **in** i: `item`, **in** it: `itLista`, **in** pre: `precio`) $\rightarrow$ res : `tupla`$\langle$`itDiccLog,itLista,nat`$\rangle$
Pre $\equiv$ {La persona dada tiene una compra en el puesto dado, en el cual se compra el item dado y el precio del item es el dado. La compra es una de las compras sin descuento de la persona(es decir, debe existir tal compra), y como tal, el iterador dado refiere a esa precisa compra}
Post $\equiv$ {El stock del item dado para el puesto dado es incrementado en 1, la venta a la que apuntaba el iterador inicial es modificada: se decrementa en uno la cantidad y el precio corresponde al precio de remover uno del item dado de la compra. El gasto en el puesto de la persona es reducido solamente por el precio del item dado. Ningun otro atributo del puesto es modificado. Si luego del hackeo, la venta sigue teniendo una cantidad no nula, res debe seguir apuntando al puesto, la venta en la lista de ventas de la persona, y el nuevo precio de la venta. Si luego del hackeo la cantidad de la compra es nula, res no debe apuntar ni al puesto ni a una compra}
Complejidad: $\mathcal{O}(\log(\text{A})+\log(\text{I}))$
Descripción: Esta operación realiza el efecto del hackeo al nivel del puesto: agrega uno del item al stock, modifica a la persona referida en la compra, reduciendo su gasto total en una unidad del item dado, modificando la compra dada o removiendola de ser necesario. Si la compra no se ve invalidada, res devuelve el acceso a la misma compra y el mismo para una posterior modificacion (es decir, otro hackeo)
Aliasing: El puesto de comida es un parametro que se pasa por referencia modificable, al igual que el iterador de la lista de ventas. El resto de los parametros de entrada no son modificables. Se devuelve una tupla de referencias modificables a la compra modificada y al puesto de la compra, en el caso de que no quede anulada la compra. En otro caso, invalida los punteros para evitar un posterior acceso

AplicarDescuento(**in** $descuento$: `nat`, **in** $nuevoPrecio$: `precio`) $\rightarrow$ res : `nat`
Pre $\equiv$ {true}
Post $\equiv \{res =_{obs} aplicarDescuento(nuevoPrecio, descuento)\}$
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el precio resultante de aplicar un descuento dado a un precio dado
Aliasing: Los parámetros de entrada se pasan por copia y res se pasa por copia

Representación

Representación de PuestoDeComida:

`puesto` **se representa con** `estr`

donde `estr` es `tupla`(*men*: `diccLog(item,tupla`⟨`precio,stock,`
`descuentos:arreglo[1..n] de nat)`⟩
, *ventasPorPersona*: `diccLog(persona,tupla`
⟨`gastoTotal:nat,ventas: secuencia(tupla`⟨`item,cantidad`⟩)⟩)
)

donde **item, precio, stock, cantidad, descuento, persona, dinero** son NAT

Invariante de representación
1) Todos los descuentos de un item, para cualquier cantidad son menores estrictos a 100.
2) La suma de todas las ventas hacia una persona, al aplicar los descuentos correspondientes, deberia dar el gasto total de dicha persona.
3) Para toda persona que tiene al menos una compra registrada en ventas, todos los items que forman parte de las compras son items que estan en el menu del puesto.

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
DescuentosValidos(e) $\wedge$
ComprasValidas(e)

Abs : estr e $\longrightarrow$ PuestoDeComida {Rep(e)}

Abs(e) $\equiv$ ($\forall$ p : PuestoDeComida)(p $=_{obs}$ e /
menu(p) = claves(e.menu) $\wedge$
($\forall$ i : item)(precio(p, i) = obtener(e.menu, i).precio
($\forall$ i : item)(stock(p, i) = obtener(e.menu, i).stock
($\forall$ i:item, c:cant)(descuento(p, i, c) =
if tam($e.descuentos$) = 0 **then**
0
else
if $c > tam(e.descuentos)$ **then**
$e.descuentos$[tam($e.descuentos$)]
else
$e.descuentos[c]$
end if
end if
$\wedge$ ($\forall$ per:persona)(ventas(p, per) =
secuenciaAMulticonjunto(obtener($e.ventasPorPersona, per$).ventas))

DescuentosValidos : estr $\longrightarrow$ bool

DescuentosValidos(e **:** $estr$**)** {
$(\forall i : item)$(def?(i,$e.menu$) $\Rightarrow_L$ $((\forall c$:cantidad$)(0 < c \leq tam(descuentos)) \Rightarrow_L descuentos[c] < 100))$ }

ComprasValidas : estr $\longrightarrow$ bool

ComprasValidas(e **:** $estr$**)** {
$(\forall\ p$:persona$)(p \in \#claves(e.ventasPorPersona) \Rightarrow_L$
$((\forall t$: tupla$\langle$item,cantidad$\rangle)$(está? (t,obtener(p,$e.ventasPorPersona$).$ventas$) $\Rightarrow_L$
$\pi_1(t) \in$ claves($e.menu$)) $\wedge$
obtener(p,$e.ventasPorPersona$).$gastoTotal$ =
sumaDeAplicarDescuentoATodo($e.menu$,obtener(p,$e.ventasPorPersona$).$ventas$))
}

sumaDeAplicarDescuentoATodo : menu $\times$ secu(tupla$\langle$item $\times$ cantidad$\rangle$) $\longrightarrow$ nat
{todo item de toda compra esta en el menu}

sumaDeAplicarDescuentoATodo(m,s) {

if vacia?(s) **then** 0
else AplicarDescuento(significado(π_1(prim(s)),e.menu)/π_2(prim(s)),
descuento(m,π_1(prim(s)),π_2(prim(s)))) + sumaDeAplicarDescuentoATodo(m,fin(s))
endif

}

descuento : menu $\times$ item $\times$ cantidad) $\longrightarrow$ nat
{todo item de toda compra esta en el menu}

descuento(m,i,c) {

if c = 0 **then** 0
else if m > tam(m.descuentos) **then** descuento(m,i,c-1)
else m.descuentos[c]
endif

}

Algoritmos

---

iCREARPUESTO(**in** p: `dicc(item, nat)`, **in** s: `dicc(item, nat)`,**in** d: `dicc(item, dicc(cant, nat))`) $\rightarrow$ res : estr

```
1:  res ← ⟨Vacío(), Vacío()⟩
2:  it ← crearIt(p)
3:  while haySiguiente(it) do
4:      precio ← significado(p,siguiente(it))
5:      stock ← significado(s,siguiente(it))
6:      descuentos ← Vacía()
7:      if definido?(d,siguiente(it)) then
8:          DescuentosaCopiar ← significado(d,siguiente(it))
9:          itDescuentos ← CrearIt(DescuentosACopiar)
10:         while haySiguiente(itDescuentos) do
11:             sign ← significado(DescuentosACopiar, siguiente(it))
12:             while longitud(descuentos) ≤ siguiente(it) do AgregarAtras(descuentos,0)
13:             end while
14:             descuentos[siguiente(it)] = sign
15:             Avanzar(itDescuentos)
16:         end while
17:         i ← 1
18:         descuentosMenores ← 0
19:         while i ≤ longitud(descuentos) do
20:             if descuentos[i] ≠ 0 then
21:                 descuentosMenores ← descuentos[i]
22:             end if
23:             descuentos[i] = descuentosMenores
24:             i ← i+1
25:         end while
26:     end if
27:     definir(res.menu, siguiente(it), ⟨precio,stock,descuentos⟩)
28:     Avanzar(it)
29: end while
30: return res
```

Complejidad: $\mathcal{O}$ ($\sum_{i' \in i}$(significado(i,i') + significado(s,i') + definido?(d,i') + significado(d,i') + $\sum_{d' \in d}$(significado(d,d') + d') + $MaximaCantConDescuento$) + #claves(p)) =0

Justificacion: Primero que nada, crear diccionarios Vacios, crear iteradores a diccionarios y avanzar dichos iteradores es $\mathcal{O}(1)$. Lo mismo asumimos de copiar y comparar numeros naturales. En el peor caso, dado una cantidad de items i, todo item tiene un descuento. Como debemos obtener tanto el precio, stock y, a la vez ver si tiene definidos los descuentos de todo item y obtenerlos, debo sumar las complejidades de significado de estos 3, y el de definido? de descuentos. Como no sabemos como estan diseñados los diccionarios dados, no podemos asegurarnos del orden de su complejidad. La variable DescuentosACopiar se pasa por referencia, por lo que no es necesario copiarlo. Finalmente, definimos los descuentos para todo item. Como trabajamos con vectores, el peor caso posible es el cual en los descuentos se incorporan en orden ascendente, y se debe reservar memoria y copiar

el vector para todo descuento nuevo, en cuyo caso se suma el pusheo de todos los indices hasta el nuevo descuento maximo. Podemos pasar los descuentos por referencia no modificable, para evitar el costo de copiar. Finalmente, para definir cada clave del nuevo diccionario, podemos usar definirRapido, ya que nos aseguramos de que no hay igualdad de claves por que los diccionarios son parámetros de entrada, así evitamos comparar la igualdad. La insercion rápida en un DiccLog es $\mathcal{O}(\log(i))$, (i es el tamaño actual del diccionario) ya que evitamos ver la igualdad y solo buscamos la ubicación del nuevo elemento. Nos queda entonces una sumatoria de $\log(i)$, con i desde 1 hasta la cantidad de claves, que sabemos que se puede acotar por la cantidad de claves(aproximacion de Stirling). Por esa razón, sumo a la complejidad previa la cantidad de items.

---

iSTOCK(**in** e : `estr`, **in** i : `item`) $\rightarrow$ res : cant

1: $res \leftarrow$ significado(e.menu,i).stock
2: return res

Complejidad: $\mathcal{O}(\log(I))$

Justificacion: Buscar el significado de una clave en e.menu tiene complejidad $\mathcal{O}(\log(I))$ siendo i la cantidad de items definidos, pues este es un diccionario logaritmico.

---

iDESCUENTO(**in** p : `estr`, **in** i : `item`, **in** c : `cant`) $\rightarrow$ res : nat

1: $diccItem \leftarrow$ significado(e.menu, i)
2: **if** tam($diccItem.descuentos$) $= 0$ **then**
3: $\quad res \leftarrow 0$
4: **else**
5: $\quad$ **if** $c >$ tam($diccItem.descuentos$) **then**
6: $\quad\quad res \leftarrow$ diccItem.descuentos[tam($diccItem.descuentos$)]
7: $\quad$ **else**
8: $\quad\quad res \leftarrow$ diccItem.descuentos[c]
9: $\quad$ **end if**
10: **end if**
11: return res
12: Complejidad: $\mathcal{O}(\log I)$

Justificacion: El algoritmo cosiste únicamente de comparaciones de naturales, acceso al tamaño de un vector y acceso a posiciones indexadas del vector, todas operaciones en $\mathcal{O}(1)$, ademas de tener que buscar el item dentro del puesto, la cual, al estar diseñado con un DiccLog, es $\mathcal{O}(\log(I))$, siendo esta la complejidad del algoritmo.

---

iGASTODE(**in/out** e : `estr`, **in** a : `persona`) $\rightarrow$ res : dinero

1: **if** $\neg$definido?(e.ventasPorPersona,a) **then**
2: $\quad$ res $\leftarrow 0$
3: **else**
4: $\quad compras \leftarrow$ significado(e.ventasPorPersona, a)
5: $\quad res \leftarrow$ compras.gastoTotal
6: **end if**
7: return res

Complejidad: $\mathcal{O}(\log(A))$

Justificacion: Primero se busca la persona de todas las personas que alguna vez realizaron una compra, lo cual es $\mathcal{O}(\log A)$, y luego se accede a un elemento de la tupla ($\mathcal{O}(1)$) Por ende, es $\mathcal{O}(\log(A))$.

Auxiliares

---

iAGREGARVENTA(**in/out** e: `estr`, **in** a: `persona`, **in** i: `item`, **in** c: `cantidad`) $\rightarrow$ res : tupla⟨itDiccLog,itLista,nat⟩

```
1:  producto ← significado(e.menu,i)
2:  producto.stock ← producto.stock - c
3:  descuento ← Descuento(p,i,c)
4:  nuevoPrecio ← producto.precio*c
5:  res ← ⟨nullptr,nullptr,0⟩
6:  nuevoPrecio ← AplicarDescuento(descuento,nuevoPrecio)
7:  if ¬(definido?(e.ventasPorPersona,a)) then
8:      definir(e.ventasPorPersona,a,⟨0,Vacia()⟩)
9:  end if
10: individuo ← significado(e.ventasPorPersona,a)
11: individuo.gastoTotal ← individuo.gastoTotal + nuevoPrecio
12: itCompra ← AgregarAtras(individuo.ventas,⟨i,c⟩)
13: if descuento = 0 then
14:     π1(res) ← puntero(e)
15:     π2(res) ← itCompra
16: end if
17: (π3(res)) ← nuevoPrecio
18: return res;
```

Complejidad: $\mathcal{O}(\log(I) + \log(A))$

Justificacion: Primero se busca el item entre todos los items del menú del puesto, lo cual es $\mathcal{O}(\log I)$. Se accede al descuento que corresponde a dicho item en la cantidad dada (O(1)) y luego se debe acceder a la persona que realiza la compra en ventasPorPersona(o definirla si no existe), pero en peor caso esto es $\mathcal{O}$(2*log(A)) = ($\mathcal{O}$(log(A)), y, ademas llama a aplicarDescuento, el cual asumimos que es $\mathcal{O}(1)$. Como el resto del algoritmo consiste en asignar variable a elementos por referencia y hacer comparaciones de naturales, aseguramos que este algoritmo es $\mathcal{O}$(log(A) + log(I))

---

iELIMINARVENTASINDESCUENTO(**in/out** e : `estr`, **in** a : `persona`, **in** i : `item`, **in** it : `itLista`, **in** p : `precioItem`) $\rightarrow$ res : tupla⟨itDiccLog,itLista,nat⟩

```
1: producto ← significado(e.menu,i)
2: producto.stock ← producto.stock+1
3: nuevaVenta ← siguiente(it)
4: nuevaVenta.cantidad ← nuevaVenta.cantidad-1
5: nuevoPrecio ← producto.precio*(nuevaVenta.cantidad)
6: res ← ⟨nullptr,nullptr,nuevoPrecio⟩
7: individuo ← significado(e.ventasPorPersona,a)
8: individuo.gastoTotal ← individuo.gastoTotal - precioItem
9: if nuevaVenta.cantidad > 0 then
10:     π1(res) ← puntero(e)
11:     π2(res) ← it
12: else
13:     EliminarSiguiente(it)
14: end if
15: return res
```

Complejidad: $\mathcal{O}(\log(I) + \log(A))$

Justificacion: Primero se busca el item en todos los items del menú del puesto, lo cual es $\mathcal{O}(\log I)$ para poder incrementar su stock. Luego se debe acceder a la persona que realiza la compra en ventasPorPersona para reducir el gasto total en el consumo de un item de la compra, el cual en peor caso es $\mathcal{O}(log(A))$ y, ademas accede a la venta del item pasada como referencia como entrada, que corresponde a la venta que queremos modificar. Como ya sabemos su posicion, el acceso a la venta es O(1). Modificamos el precio de la venta (que ya sabemos que no va a tener descuento) y la cantidad del item consumido, y comparamos: si la nueva cantidad es 0, eliminamos la venta de nuestra lista de ventas e impedimos el acceso a la lista de ventas devolviendo nullpointers. Si no es 0, devolvemos la referencia moodificada, con el nuevo precio. Todo salvo buscar el item y la persona en el puesto es O(1), por lo que la complejidad es $\mathcal{O}(\log(I) + \log(A))$

iAplicarDescuento(**in** *descuento* : `nat`, **in** *nuevoPrecio* : `precio`) $\rightarrow$ res : nat

```
1: res ← (nuevoPrecio − ((nuevoPrecio×descuento)÷100))
```

Se asume que la division es flotante y se realiza en O(1), asi siempre aplicar un 0 % de descuento mantiene el valor del producto y se cumple la cota de complejidad de vender.

4. Módulo ColaDePrioridad(α)

Se explica con: Cola De Prioridad

géneros: `ColaPrioridad`

Interfaz

Usa: bool, α

Parametros formales

generos α

Copiar(**in** a: α) $\rightarrow$ res : α
Pre $\equiv$ {True}
Post $\equiv$ {$res =_o bs\ Copiar(a)$}
Complejidad: O(copy(a))
Descripción: Se devuelve una copia del elemento
Aliasing: Crea una copia del elemento

$>_\alpha$(**in** $a1$: α, **in** $a2$: α) $\rightarrow$ res : `bool`
Pre $\equiv$ {True}
Post $\equiv$ {$res =_o bs\ a1 >_\alpha a2$}
Complejidad: O($(>_\alpha)(a1, a2)$)
Descripción: Los parámetros se pasan por referencia no modificable.
Aliasing: Devuelve true si y solo si, dada una relacion de orden, el primer parámetro es mayor al segundo

Operaciones:

Vacia?(**in** c: `colaPrioridad(`α`)`) $\rightarrow$ res : `bool`
Pre $\equiv$ {True}
Post $\equiv$ {$res =_{obs}$ Vacia?(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve true si y solo si la cola de prioridad dada no tiene elementos encolados
Aliasing: Los parámetros de entrada son pasados por referencia no modificable

Proximo(**in** c: `colaPrioridad(`α`)`) $\rightarrow$ res : α
Pre $\equiv$ {$\neg$ Vacía?(c)}
Post $\equiv$ {$res =_{obs}$ proximo(c)}
Complejidad: $\mathcal{O}(1)$
Descripción: Devuelve el siguiente elemento a desencolar de una cola de prioridad no vacia
Aliasing: Res es modificable si y solo si c lo es

Desencolar(**in/out** c: `colaPrioridad(`α`)`)
Pre $\equiv$ {$\neg$ Vacía?(c)}
Post $\equiv$ {$res =_{obs}$ desencolar(c)}
Complejidad: $\mathcal{O}$(log(tamaño(c)))
Descripción: Desencola el elemento de más prioridad de la cola, asumiendo que se da una cola no vacia
Aliasing: c entra y sale como referencia modificable

ObtenerPrioridad(**in/out** c: `colaPrioridad(`α`)`)
Pre $\equiv$ {$\neg$ Vacía?(c)}
Post $\equiv$ {$res =_{obs} c.masPrioridad$}

Complejidad: $\mathcal{O}(1)$
Descripción: Obtiene el elemento de mayor prioridad de una cola
Aliasing: c entra y sale como referencia modificable

VACIA() $\rightarrow$ *res* : `colaPrioridad(`α`)`
Pre $\equiv$ {True}
Post $\equiv$ {*res* $=_{obs}$ Vacia()}
Complejidad: $\mathcal{O}(1)$
Descripción: Crea una cola de prioridad sin elementos
Aliasing: Se devuelve una referencia modificable a res

ENCOLAR(**in/out** c : `colaPrioridad(`α`)`, **in** $a : \alpha$) $\rightarrow$ *res* : `puntero(colaPrioridad(`α`))`
Pre $\equiv$ {c $=_o$ *bs* c_0}
Post$\equiv$ {$c =_{obs}$ encolar(c, a) $\wedge$ *res.raiz $=_o$ *bs* a}
Complejidad: $\mathcal{O}$(log(tamaño(c))
Descripción: Encola el elemento a según su prioridad con respecto al resto de los elementos de la cola, de haberlos. Caso contrario, se inserta colo elemento de más prioridad
Aliasing: c y a se pasan como referencias, donde c es modificable, a no lo es, y res se pasa como referencia modificable

Auxiliares
SIFTUP(**in/out** c : `colaPrioridad(`α`)`,**in/out** a : : ArbolConPadre(α)) $\rightarrow$ *res* : `itColaPrioridad`
Pre $\equiv$ {a es un elemento de la cola de prioridad, y todos sus atributos coinciden con los del nodo de a en c}
Post $\equiv$ {La cola es una permutacion de la pasada por entrada, en la que se cumple el invariante de representacion de un heap}
Complejidad: $\mathcal{O}(\prod$n $\in$ caminoHastaRaizDesdeA($>_\alpha$(n.raiz,n.padre.raiz)))
Descripción: Swapea la posición de A con la de su padre, mientras A tenga padre y A tenga mayor prioridad a la de este
Aliasing: Los parámetros de entrada se pasan por referencia modificable y devuelve una referencia modificable a la cola de prioridad

SIFTDOWN(**in/out** c : `colaPrioridad(`α`)`,**in/out** a : : arbolConPadre(α))
Pre $\equiv$ {a es un elemento de la cola de prioridad, y todos sus atributos coinciden con los del nodo de a en c}
Post $\equiv$ {La cola es una permutacion de la pasada por entrada, en la que se cumple el invariante de representacion de un heap}
Complejidad: $\mathcal{O}(\prod$n $\in$ maxCaminoHastaPosicionDesdeA($>_\alpha$(n.raiz,*(n.der).raiz) + $>_\alpha$(n.raiz,*(n.izq).raiz) + $>_\alpha$(*(n.izq).raiz,*(n.der).raiz))
Descripción: Swapea la posicion de a con la del mayor de sus hijos, mientras A tenga hijos y alguno de ellos tenga mayor prioridad
Aliasing: Los parámetros de entrada se pasan por referencia modificable, y devuelve una referencia modificable a una posicion de la cola de prioridad

SWAP(**in/out** c : `colaPrioridad(`α`)`,**in/out** $Mayor$: `arbolConPadre(`α`)`, **in/out** $Menor$: `arbolConPadre(`α`)`) $\rightarrow$ *res* : `tupla` $\langle$`arbolConPadre(`α`),arbolConPadre(`α`)`$\rangle$
Pre $\equiv$ {Mayor y Menor son elementos de la cola de prioridad, que se asocian de forma que Mayor es padre de Menor y uno de los hijos de Mayor es menor}
Post $\equiv$ {La cola es una permutacion de la pasada por entrada, el la que se cumple que, luego del swap, Menor es padre de Mayor, Mayor es alguno de los hijos de menor, los hijos de menor son ahora los de mayor y viceversa. Ningun otro nodo de la cola de prioridad se ve afectado. Res es igual a la tupla (NuevoMayor,NuevoMenor), que revierte los ordenes de los parametros de la entrada}
Complejidad: $\mathcal{O}(1)$
Descripción: Swapea la posicion de a con la del mayor de sus hijos, mientras A tenga hijos y alguno de ellos tenga mayor prioridad
Aliasing: Los parámetros de entrada se pasan por referencia modificable, y res es una

referencia modificable

Nota: el punto de este algoritmo es que los punteros de la cola de prioridad mantengan la referencia al nodo con el valor con los que fueron inicializados. Puede estar mal diseñado el algoritmo, pero el objetivo del algoritmo es ese. El unico proposito de este algoritmo es para que vender y hackeo mantengan la consistencia en los punteros a los nodos de el que mas gasto, asi, una vez que se aumenta o disminuye la prioridad de un individuo, el puntero al que apunta todo individuo en el puesto o cola de prioridad siga apuntando al mismo.

Representación

Representación de ColaPrioridad(α):

`colaPrioridad(`α`)` **se representa con** `estr`

donde `estr` es `tupla`(*másPrioridad*: `puntero(ArbolConPadre(`α`))` , *tamaño*: `nat`)

donde ArbolConPadre(α) es tupla(
padre : puntero(ArbolConPadre(α)),
izq : puntero(ArbolConPadre(α)),
der : puntero(ArbolConPadre(α))

Invariante de Representacion
Dado que el arbol no es null:
1) El elemento de mayor prioridad no tiene padre
2) Tanto el hijo derecho como izquierdo de un nodo tienen como padre a dicho nodo. El hijo derecho, si no es nullptr, no puede apuntar a lo mismo que el hijo izquierdo y viceversa
3) La raiz del arbol tiene mayor prioridad, dada la relacion de orden, que sus hijos derecho e izquierdo, si tales existen
4) El arbol está balanceado
5) 2), 3) y 5) se cumplen recursivamente para los subarboles derecho e izquierdo
6) El arbol es izquierdista, es decir, llena los niveles de izquierda a derecha
7)El tamaño coincide con la cantidad de nodos del arbol 8)No se producen repeticiones de punteros(excepto de padre) en el arbol, es decir, cada puntero a hijo derecho e izquierdo de un nodo no se repite

Rep : estr $\longrightarrow$ bool

Rep(e) $\equiv$ true $\Longleftrightarrow$
másPrioridad = nullptr $\vee_L$ (**RaizSinPadre**(e) $\wedge$
PadreDeHijos($*(e.raiz)$) $\wedge$
balanceado($*(e.raiz)$) $\wedge$
PadreMayorPrioridad($*(e.raiz)$) $\wedge_L$ **HijosCumplen**($*(e.raiz)$) $\wedge$
izquierdista($*(e.raiz)$)) $\wedge$ **tamañoCoincide**($e.raiz$,**e.tamaño**) $\wedge$
sinPunterosRepetidos($e.raiz$,**e.tamaño**)

Abs : estr e $\longrightarrow$ ColaPrioridad(α) {Rep(e)}

Abs(e) $\equiv$ ($\forall$ c : colaPrioridad(α))(c $=_{obs}$ e /
Vacia?(c) = (e.masPrioridad = nullptr) $\wedge$
$\neg$ Vacia?(c) $\Rightarrow_L$
(proximo(c) = *(e.masPrioridad).raiz $\wedge$
Desencolar(c) = e.sinMaximo)

No especificamos sinMaximo, pero damos a entender que elimina el elemento de mayor prioridad de la estructura, de manera que el 2do elemento de mayor prioridad (si tal existe) ahora sea el de mayor prioridad, manteniendo el invariante de representacion. Si el elemento que borramos era el único, e.masPrioridad será un nullptr

RaizSinPadre : estr $\longrightarrow$ bool

RaizSinPadre(e **:** $estr$) $\rangle$)) {
*(e.masPrioridad).padre = nullptr; }

PadreDeHijos : ArbolConPadre(α) $\longrightarrow$ bool

PadreDeHijos(a **: ArbolConPadre**(α)) {
($\forall(aIzq, aDer : arbolConPadre(\alpha)$))((aIzq.padre $\neq$ nullptr $\wedge_L$ *(aIzq.padre) = a $\Longleftrightarrow$
a.izq $\neq$ nullptr $\wedge_L$ *(a.izq) = aIzq) $\wedge$ (aDer.padre $\neq$ nullptr $\wedge_L$ *(aDer.padre) = a $\Longleftrightarrow$
a.der $\neq$ nullptr $\wedge_L$ *(a.der) = aDer) }

balanceado : ArbolConPadre(α) $\longrightarrow$ bool

balanceado(a **: ArbolConPadre**(α)) { altura(*(a.izq)) == altura(*(a.der)) $\vee$ altura(*(a.izq)) == altura(*(a.der))+1 $\vee$ altura(*(a.izq)) == altura(*(a.der))-1 $\wedge$ (a.der $\neq$ nullptr $\Rightarrow_L$ balanceado(*(a.der)) $\wedge$ (a.izq $\neq$ nullptr $\Rightarrow_L$ balanceado(*(a.izq)))) }

altura : ArbolConPadre(α) $\longrightarrow$ nat

altura(a **: ArbolConPadre**(α)) {
if a = NULL **then** 0
else (1+maxaltura(*($a.izq$)),altura(*($a.der$)))
endif
}

PadreMayorPrioridad : ArbolConPadre(α) $\longrightarrow$ bool

PadreMayorPrioridad(a **: ArbolConPadre**(α)) {
a.izq $\neq$ *nullptr* $\Rightarrow_L$ a.raiz $>_\alpha$ *(a.izq).raiz $\wedge$ a.der $\neq$ nullptr $\Rightarrow_L$ a.raiz $>_\alpha$ *(a.der).raiz }

HijosCumplen : ArbolConPadre(α) $\longrightarrow$ bool

HijosCumplen(a : ArbolConPadre(α)) {
a.izq $\neq$ nullptr $\Rightarrow_L$ (PadreDeHijos(*(a.izq)) $\wedge$ PadreMayorPrioridad(*(a.izq)) $\wedge$ HijosCumplen(*(a.izq)) $\wedge$ a.der $\neq$ nullptr $\Rightarrow_L$ (PadreDeHijos(*(a.der)) $\wedge$ PadreMayorPrioridad(*(a.der)) $\wedge$ HijosCumplen(*(a.der))) }

izquierdista : ArbolConPadre(α) $\longrightarrow$ bool

izquierdista(a : ArbolConPadre(α)) {
a = Null $\vee_L$ Nivel(*(a.izq)) $\geq$ Nivel(*(a.der)) $\wedge$ a.izq $\neq$ nullptr $\Rightarrow_L$ izquierdista(*(a.izq)) $\wedge$ a.der $\neq$ nullptr $\Rightarrow_L$ izquierdista(*(a.der)) }

tamañoCoincide : puntero(ArbolConPadre(α)) $\times$ nat $\longrightarrow$ bool

tamañoCoincide(a : puntero(ArbolConPadre(α)), n:nat) {
n = a.tamaño }

tamaño : puntero(ArbolConPadre(α)) $\longrightarrow$ nat

tamaño(a : puntero(ArbolConPadre(α))) {
if a = nullptr **then** 0
else 1+tamaño(*a.izq)+tamaño(*a.der)
endif }

sinPunterosRepetidos : puntero(ArbolConPadre(α)) $\times$ nat $\longrightarrow$ bool

sinPunterosRepetidos(pa : puntero(ArbolConPadre(α)), n:nat) {
#PunterosDelArbol(pa) = n }

PunterosDelArbol : puntero(ArbolConPadre(α)) $\times$ nat $\longrightarrow$ bool

sinPunterosRepetidos(pa : puntero(ArbolConPadre(α)), n:nat) {
if pa = nullptr **then** $\emptyset$
else Ag(pa, PunterosDelArbol(*pa.der) $\cup$ PunterosDelArbol(*pa.izq))
endif
}

Algoritmos

---

iVacia?(**in** e : `estr`) $\rightarrow$ res : bool

1: res $\leftarrow$ e.masPrioridad = nullptr
2: return res

Complejidad: $\mathcal{O}(1)$
Justificacion: Simplemente compara un puntero al que se tiene acceso directo con nullptr, comparación que se asume que tarda tiempo constante

---

---

iObtenerPrioridad(**in** e : `estr`) $\rightarrow$ res : bool

1: $res \leftarrow e.masPrioridad$
2: return res

Complejidad: $\mathcal{O}(1)$
Justificacion: Devuelve el puntero de mas prioridad

---

iProximo(**in** e : `estr`) $\rightarrow$ res : α

1: $res \leftarrow *(e.masPrioridad).raiz$
2: return res

Complejidad: $\mathcal{O}(1)$

Justificacion: Buscar la raiz del elemento con mayor prioridad, al cual, asumiendo que esta definido, se tiene acceso en $\mathcal{O}(1)$, ya que no necesito realizar una busqueda para encontrarlo.

iDesencolar(**in/out** e : `estr`)

```
1:  tam ← e.tamaño
2:  if tam = 1 then
3:      e.masPrioridad ← nullptr
4:  else
5:      actual ← *(e.masPrioridad)
6:      temp ← tamaño
7:      listaBinario ← Vacia()
8:      while temp>0 do
9:          AgregarAdelante(listaBinario,temp mod 2)
10:         temp ← temp div 2
11:     end while
12:     it ←(Siguiente(CrearIt(listaBinario)))
13:     while haySiguiente(haySiguiente((it)) do
14:         if siguiente(it) = 1 then
15:             actual ← *(actual.der)
16:         else
17:             actual ← *(actual.izq)
18:         end if
19:     end while
20:     if siguiente(it) = 1 then
21:         actual ← *(actual.der)
22:         *(actual.padre).der ← nullptr
23:     else
24:         actual ← *(actual.izq)
25:         *(actual.padre).izq ← nullptr
26:     end if
27:     actual.padre ← nullptr
28:     actual.izq ← *(e.masPrioridad).izq
29:     actual.der ← *(e.masPrioridad).der
30:     *(e.masPrioridad).izq ← nullptr
31:     *(e.masPrioridad).der ← nullptr
32:     if actual.der ≠ nullptr then
33:         *(*(actual.der).padre) ← actual
34:     end if
35:     if actual.izq ≠ nullptr then
36:         *(*(actual.izq).padre) ← actual
37:     end if
38:     *(e.masPrioridad) ← actual
39:     actual2 ← *(e.masPrioridad)
40:     siftDown(e,actual2)
41: end if
42: e.tamaño ← e.tamaño-1
```

Complejidad:$\mathcal{O}$(log(e.tamaño))

Justificacion: Primero se hace una representación binaria del tamaño del heap, lo cual es $\mathcal{O}$(log(e.tamaño)), asumiendo que div y mod son O(1). Luego se busca el ultimo elemento del heap que, dado por su invariante de representacion, se puede obtener mediante la representacion binaria del tamaño del heap, y recorro las ramas hasta llegar a este. Como se sabe que camino lleva al último nodo, solo se recorren $\mathcal{O}$(log(e.tamaño)) elementos. Se reemplaza el valor del elemento de mayor prioridad con el ultimo que, como se pasan por referencia, es $\mathcal{O}(1)$. Finalmente, mientras la raiz del otrora último elemento sea menor al de alguno de los hijos de su posicion actual, reemplaza la posicion por la del mayor de sus hijos. En peor caso, se tiene que hacer reemplazos hasta una hoja del heap, por lo que se producen a lo sumo $\mathcal{O}$(log(e.tamaño)) iteraciones. Por ende, al sumar las complejidades, la complejidad de este algoritmo es $\mathcal{O}$(log(e.tamaño))

iVacia() → *res* : estr

1: *res* ← ⟨nullptr,0⟩

Complejidad: $\mathcal{O}(1)$

Justificacion: Se hace una asignacion de un puntero nulo y un natural, juntos en una tupla. Como la copia de naturales es $\mathcal{O}(1)$ y asignar un puntero a nullptr se asume $\mathcal{O}(1)$, esto es $\mathcal{O}(1)$

---

iEncolar(**in/out** e : `estr`, **in** $a : \alpha$)→ *res* : itColaPrioridad

```
1: tam ← e.tamaño
2: if tam = 0 then
3:     *(e.masPrioridad) ← ⟨a,nullptr,nullptr,nullptr⟩
4: else
5:     actual ← *(e.masPrioridad)
6:     temp ← tamaño+1
7:     listaBinario ← Vacia()
8:     while temp>0 do
9:         AgregarAdelante(listaBinario,temp mod 2)
10:        temp ← temp div 2
11:    end while
12:    it ← (Siguiente(CrearIt(listaBinario)))
13:    while haySiguiente(haySiguiente((it)) do
14:        if siguiente(it) = 1 then
15:            actual ← *(actual.der)
16:        else
17:            actual ← *(actual.izq)
18:        end if
19:    end while
20:    if siguiente(it) = 1 then
21:        *(actual.der) ← ⟨puntero(actual),nullptr,nullptr,a⟩
22:        actual ← actual.der*
23:    else
24:        *(actual.izq) ← ⟨puntero(actual),nullptr,nullptr,a⟩
25:        actual ← *(actual.der)
26:    end if
27:    actual ← siftUp(e,actual)
28: end if
29: e.tamaño ← e.tamaño+1
30: res ← puntero(actual)
31: return res
```

Complejidad:$\mathcal{O}$(log(e.tamaño))

Justificacion: Se asume que operaciones de enteros como division y modulo son O(1). Primero se crea una lista enlazada con la representacion binaria del tamaño + 1, lo cual es O(log(e.tamaño+1)), porque en estas listas agregar elementos al inicio es O(1). Luego se recorre, dada esta representacion binaria, los nodos del heap hasta el padre del siguiente nodo, lo cual tambien es O(log(e.tamaño)). Finalmente, se hace un llamado a siftUp, el cual es O(log(e.tamaño)) en peor caso. Como las raices, para tipos no primitivos, se pasan por referencia, no hay complejidad agregada por copiar elementos.

Auxiliares

---

iSiftUp(**in/out** e: `estr`, **in/out** a: `arbolConPadre(`α`)`)$\rightarrow$ res : puntero(arbolConPadre(α))

act $\leftarrow$ a

while $act \neq *(e.masPrioridad) \wedge act.raiz >_\alpha act.padre * .raiz$ **do**

act $\leftarrow \pi_1$(swap(*(act.padre),act))

end while

res $\leftarrow$ puntero(act)

return res

Complejidad:O($\prod_{n \in caminoHastaRaizDesdeA}(>_\alpha (n.raiz, n.padre.raiz))$

Justificacion: Para toda iteracion del ciclo, se debe comparar el valor de la raiz del actual con el de su padre. En el caso mas frecuente de llamado de esta funcion, se parte de una hoja, en cuyo peor caso se llega a la raiz, (log(e.tamaño) iteraciones),y las comparaciones son O(1), por lo que esto se resuma a O(log(e.tamaño)) en dichos casos

---

---

iSiftDown(**in/out** c: `ColaDePrioridad(`α`)`, **in/out** a: `ArbolConPadre(`α`)`)

act $\leftarrow a$

while act.der $\neq nullptr$ $wedge_L$ $*(a.der).raiz >_\alpha$ a.raiz $\vee$ a.izq $\neq nullptr$ $\wedge_L$ $*(a.izq).raiz >_\alpha$ a.raiz **do**

act $\leftarrow$ a

if $act.izq = nullptr \vee *(act.der).raiz >_\alpha *(act.izq).raiz$ **then**

act $\leftarrow \pi_2$(swap(act, act.der*.raiz))

else

if $act.der = nullptr \vee *(act.izq).raiz >_\alpha *(act.der).raiz$ **then**

act $\leftarrow \pi_2$(swap(act, *(act.izq).raiz))

end if

end if

end while

Complejidad:$\mathcal{O}(\prod_{n \in maxCaminohastaPosicionDesdeA}(>_\alpha$(n.raiz,*(n.der).raiz) +
$>_\alpha$(n.raiz,*(n.izq).raiz) +
$>_\alpha$(*(n.izq).raiz,*(n.der).raiz)))

Justificacion:Para toda iteracion del ciclo, se debe comparar el valor de la raiz del actual con el de su ambos de sus hijos, y los hijos entre si, si tales existen. En el caso mas frecuente de llamado de esta funcion, se parte de la raiz, en cuyo peor caso debemos reemplazar hasta la hoja (log(e.tamaño) iteraciones),y las comparaciones son O(1), por lo que esto se resuma a O(log(e.tamaño)) en esos casos.

---

iSwap(**in/out** c: `ColaDePrioridad(`α`)`, **in/out** $Mayor$: `ArbolConPadre(`α`)`,**in/out** $Menor$: `ArbolConPadre(`α`)`)

if Mayor.padre $\neq$ nullptr **then**
  if (*(*Mayor.padre).izq) = (Mayor) **then**
    ((Mayor.padre).izq) $\leftarrow$ Menor
  else
    ((Mayor.padre).der) $\leftarrow$ Menor
  end if
  *(Mayor.padre) $\leftarrow$ Menor
  Menor.padre $\leftarrow$ Mayor.padre
end if
if Menor.izq $\neq$ nullptr **then**
  ((Menor.izq).padre) $\leftarrow$ Mayor
end if
if Menor.der $\neq$ nullptr **then**
  ((Menor.der).padre) $\leftarrow$ Mayor
  if Mayor.izq $\neq$ nullptr $\wedge$ *(Mayor.izq) = Menor **then**
    temp $\leftarrow$ Menor.izq
    Menor.izq $\leftarrow$ Mayor.izq
    Mayor.izq $\leftarrow$ temp
    Mayor.der $\leftarrow$ Menor.der
    *(Menor.der) $\leftarrow$ Mayor
    if Menor.izq $\neq$ nullptr **then**
      ((Menor.izq).padre) = Menor
    end if
  else
    if Mayor.der $\neq$ nullptr $\wedge$ *(Mayor.der) = Menor **then**
      temp $\leftarrow$ Menor.der
      Menor.der $\leftarrow$ Mayor.der
      Mayor.der $\leftarrow$ temp
      Mayor.izq $\leftarrow$ Menor.izq
      *(Menor.izq) $\leftarrow$ Mayor
      if Menor.der $\neq$ nullptr **then**
        ((Menor.der).padre) = Menor
      end if
    end if
    res $\leftarrow$ $\langle$Menor,Mayor$\rangle$

Complejidad:$\mathcal{O}(1)$

Justificacion: Toda operacion del algoritmo trata sobre asignacion y comparación de punteros, lo cual se asume O(1).